We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

4,800
Open access books available

122,000
International authors and editors

135M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

10

PPCea: A Domain-Specific Language for Programmable Parameter Control in Evolutionary Algorithms

Shih-Hsi Liu[1], Marjan Mernik[2], Mohammed Zubair[1],
Matej Črepinšek[2] and Barrett R. Bryant[3]
[1]*California State University, Fresno,*
[2]*University of Maribor,*
[3]*University of Alabama at Birmingham,*
[1,3]*USA*
[2]*Slovenia*

1. Introduction

An Evolutionary Algorithm (EA) is a meta-heuristic and stochastic optimization search process that mimics Darwinian evolution theory and Mendel's Genetics. Each process facilitates (a) population(s) evolve into fittest and/or convergence by setting parameters of selection, mutation, crossover, population resizing, and/or many other variant operators. However, due to two primary identified factors, EAs are still a challenging research topic: (1) Value choices/ranges for parameters (i.e., parameter settings) will greatly influence the evolution performance of a search process in terms of fittest and/or convergence; and (2) Parameter settings that are good for one fitness function do not guarantee the same evolution performance of another fitness function. Namely, parameter settings are function-specific. Different functions may have various characteristics that request specific attention. In order to better organize and overcome the parameter setting problem, Eiben et al. have classified parameter settings into parameter tuning and parameter control (Eiben et al., 1999): Parameter tuning determines parameter values before a search process begins while parameter control changes parameter values during a search process. More specifically, parameter control adjusts parameters on-the-fly using three different approaches: (1) Deterministic approach alters parameters based on certain pre-determined rules or formulae; (2) Adaptive approach strategically adjusts parameter values based on the feedbacks of a search process. Such feedbacks could be fitness, diversity, distance, among others; and (3) Self-adaptive approach encodes parameters to be adapted and evolves them along with a search process. Yet, even with such a classification, to our best knowledge there is no existing tool to assist researchers with conducting experiments of parameter settings with ease. Namely, researchers need to find out appropriate places out of thousand lines of EA source code to introduce and update specific parameters (including feedbacks) as well as formulae and adaptive strategies. Additionally, a number of revisions for EA source code will be also required for different kinds of experiments. To EA experimenters, such endeavor is time consuming and error prone. To EA developers, complex and tangling source code, resulted from different

parameter and strategy introductions, may also cause inflexibility for further extension and inevitability of faulty EA source code. In order to solve the aforementioned problems, a programmable approach, called PPCea (Programmable Parameter Control for Evolutionary Algorithms) (Liu et al., 2004), is presented in this book chapter.

PPCea is a Domain-Specific Language (DSL) (Mernik et al., 2005) for EAs. It uplifts the abstraction layer to a higher (i.e., domain-specific) level and introduces domain-specific notations (e.g., parameters and statements) as well as common linguistic elements. Namely, the implementation details of Genetic Algorithms (GAs) and Evolution Strategies (ESs) are encapsulated and hidden so that EA experimenters are able to experiment with evolutionary algorithms and obtain statistical results by programming a few PPCea statements. Additionally, the flexible programming fashion also enhances the possibility of reproducing existing EA experiments in a simpler PPCea source code and likely introducing new experiments to facilitate even better optimization search or faster convergence.

For EA experimenters, the first part of this book chapter introduces PPCea with examples to demonstrate PPCea's capabilities and usability. Famous existing parameter tuning and parameter control examples are reproduced using PPCea (e.g., Fogarty's formula (Fogarty, 1989), PROFIGA (Eiben et al., 2004), and 1/5 success rule (Bäck & Schwefel, 1995)). Additionally, new examples are also demonstrated to show the flexibility of PPCea. For example, introducing new metrics as feedbacks for parameter control and adaptively switching among different operators during an evolutionary process can be done with ease.

For EA developers, design and implementation of PPCea are covered in the second part of the book chapter. In this part, DSL patterns and design patterns are utilized. Coding and/or UML examples are presented and discussed to show how such patterns lessen the extension problems during development and maintenance phases. Software metrics are also measured to prove the effectiveness of design patterns for modularization and extension. In summary, PPCea is a domain-specific tool that is "win-win" to both EA experimenters and developers: For EA experimenters, the programmable fashion and high level abstraction allow EA users to conduct EA experiments in a productive manner. For EA developers, the design and implementation of PPCea allow evolutionary algorithms, operators, algorithms of operators (i.e., strategies), and parameters to be introduced or revised painlessly.

The book chapter is organized as follows. By using grammars, code snippets, and UML diagrams, Sections 2 and 3 respectively introduce PPCea from the perspectives of experimenters and developers. Section 4 discusses related work on parameter settings in Evolutionary Algorithms. PPCea's capabilities, limitations, and future directions are concluded in Section 5.

2. PPCea: A Painless Problem Curer for EA users

Because of the meta-heuristic and stochastic characteristics towards searching optimization, experimenters or users of EAs are inevitably requested to perform a sufficient number of experiments. Needless to say, there are numerous combinations and scopes of domain-specific parameters (e.g., mutation rate, crossover rate, and selection pressure) need to be tuned or controlled so that fittest and/or convergence can be discovered. A primary objective of PPCea is to become a problem curer for EA users/experimenters to conduct experiments painlessly. We first introduce PPCea through a number of examples categorized by Eiben et al.'s classification suggestions. The grammar of PPCea is appended at the end of the chapter for interested readers.

2.1 PPCea for parameter tuning

Parameter tuning is an approach for EA experiments classified in (Eiben et al., 1999). Such a kind of experiments determines the parameters of an evolutionary process before it runs and will not change the parameter values during the process. Many of the existing EAs are classified into this category. Per their endeavors, common guidelines for setting mutation and crossover rates in GAs are as follows: mutation rate (*pm*) ≒ 1/(the bit length of an individual in genetic algorithms) and crossover rate (*pc*) ≒ 0.75~0.95. PPCea can reproduce such experiments easily. Figure 1 shows that twenty experiments of Ackley's function from (Yao et al., 1999) with different parameter tuning settings are defined using Grefenstette's guideline (Grefenstette, 1986). Also, if one does not want to reset different values for *pm, pc,* or any domain-specific parameters for each experiment, formulae may be defined to adjust the values of such parameters as seen in the italic part of Figure 1, where the *if*-statement within the *while*-statement adjusts *pm* every 5 experiments. The experimental results of three reproduced parameter tuning-based experiments are available at (Liu, 2010).

```
genetic
    readfile weightF10.txt; //load coeff. of Ackley's function from Yao et al., 1999
    Function := 10; //load Ackley's function from Yao et al., 1999
    Round := 20; //number of experiments
    Maxgen := 1000; //maximum generation of an evolutionary process
    pm := 0.001; //set mutation rate
    pc := 0.95; //set crossover rate
    r := 0;
    while ( r < Round ) do
        init; //initialize population
        callGA; //invoke an evolutionary process of GA
        if (( r % 5) == 0) then
            pm := pm + 0.001 //change pm every 5 experiments
        fi;
        r := r + 1
    end;
    writeresult //output the experimental results to text and Excel files
end genetic
```

Fig. 1. Parameter tuning using PPCea

2.2 PPCea for deterministic parameter control

```
genetic
//skip initializing Round, Maxgen, Popsize, Epoch, pm, alpha, beta, gamma, length, r, g
    while ( r < Round ) do
        init; //initialize population
        while ( g < Maxgen ) do
            callGA; //invoke an evolutionary process of GA
            pm := sqrt(alpha / beta) * exp((0 - gamma)*g/2) / (Popsize / length);
        // the above formula is from Hessen & Manner
        // pm := 1 / (2+(( length-2 )/Maxgen)* g )
        // the above formula is from Bäck & Schütz
            g := g + Epoch // Generation stride for parameter control adaptation
        end;
        r := r + 1
    end;
    writeresult //output the experimental results to text and Excel files
end genetic
```

Fig. 2. Deterministic parameter control using PPCea

An important advantage of PPCea over other EA frameworks or software is its capability of performing parameter changes on-the-fly through a programmable fashion. Deterministic parameter control is an approach that defines how to change parameters during an evolutionary process using formulae. Fogarty (Fogarty, 1989) proposed one of the earliest deterministic approaches that adjusts mutation rate to a smaller value along with generations in order to tend from exploration towards exploitation. Liu et al. has published the experimental results of five unimodal and seven multimodal functions using Fogarty's mutation rate formula in (Liu et al., 2009). Figure 2 reproduces (Hesser & Männer, 1991)'s and (Bäck & Schütz, 1996)'s mutation formulae to show that PPCea is capable of representing more sophisticated cases.

2.3 PPCea for adaptive parameter control

Different from deterministic parameter control that does not interact with the evolutionary process that it controls, adaptive parameter control utilizes the analysis results from the evolutionary process and then determines which directions the evolutionary process may move forward by changing the parameters of associated operators. PPCea has reproduced 1/5 success rule (Bäck & Schwefel, 1995) and population resizing (Smith & Smuda, 1995) and introduced an entropy-driven approach (Liu et al., 2009) to adapt an evolutionary process. Figure 3 shows that PROFIGA (Eiben et al., 2004) is reproduced by PPCea.

PROFIGA is a GA that utilizes population resizing to balance between exploration and exploitation. As seen in the first *if*-statement in the figure, if the best fitness is improved, then population size will be increased proportionally so that more exploration can be promoted. Similarly, if the evolutionary process is not improved every *kgen* generations, the population size will be proportionally increased using the same factor (*growFactorX*). The second *if*-statement performs such an objective. Note that *growFactorX* is a negative value so that the formula within the second *resize* uses subtract operator. Lastly, the last *if*-statement shows that if neither the first nor the second conditions hold, the evolutionary process will tend to exploitation by shrinking the population size.

Of course, PPCea is not almighty. For example, GAVaPS (Arabas et al., 1994) and APGA (Bäck et al., 2000) perform population resizing based on aging concept. Such algorithms cannot be reproduced by current PPCea due to absence of age in individuals. Yet, once age is introduced along with associated operators, PPCea is capable of performing GAVaPS and APGA without a doubt. Similarly, parameter-less GA (Harik & Lubo, 1999) introduces a number of populations with different sizes to compete with each other. Because PPCea currently does not introduce multi-populations, reproducing parameter-less GA is also questionable. Because the design and implementation of PPCea facilitate extension and evolution, new algorithms like GAVaPS, APGA, parameter-less GA, and other EAs may be introduced with ease. More discussions on how to utilize such design and implementation advantages to introduce new algorithms will be covered in Section 3.

2.4 PPCea for adaptive operator control

Adapting parameters on-the-fly is not new in EAs. What about adapting operators on-the-fly? Adapting operators may be classified into three categories:

1. Operator adaptation is delegated to parameter control. Such an adaptation is done by adjusting parameters associated to specific operators. For example, 1/5 success rule utilizes mutation success rate to determine if mutation rate needs to be tuned up or

```
genetic
// initialize all needed parameters. bestImproved and noImprovedForLong are false
    init;
    initBest := Best; // best fitness from initial population
    nextBest := initBest;
    while (g < Maxgen) do
        currBest := nextBest; // best fitness from the current population
        callGA; // invoke an evolutionary process
        nextBest := Best;  // best fitness from the population of next generation
        growFactorX := factor * (Maxgen – g )* Popsize * (nextBest – currBest) /
        initBest;
        if ((nextBest – currBest) > 0) then //best fitness improved
            resize(Popsize * (1 + growFactorX));
            bestImproved := true
        fi;
        if ((nextBest – currBest) < 0 ) then //best fitness not improved for kgen
            i := i + 1;
            if ( i == kgen ) then
                i := 0
            fi;
            resize(Popsize * (1 – growFactorX)); //Popsize increase
            noImprovedForLong := true
        fi;
        if ((bestImproved != true) && (noImprovedForLong != true)) then
            resize(Popsize * (1 – 0.05))
        fi;
        g := g + Epoch;
    end
end genetic
```

Fig. 3. PROFIGA reproduction using PPCea

down. Similarly, selection pressure assists in adjusting the performance of selection operator in terms of fitting offspring. Usually adaptation in this category is classified as parameter control (Eiben et al., 1999);

2. Instead of focusing on the effectiveness of a specific operator using parameter control, an evolutionary process may switch among different operators based on certain real-time feedbacks. For example, Ursem (Ursem, 2002) introduced Diversity-Guided Evolutionary Algorithm (DGEA) that splits an evolutionary process into exploration and exploitation modes based on diversity. Under exploitation mode, recombination and selection are active. Otherwise, mutation is in charge;
3. Adaptation can be also done by switching among different variants of the same type of operators (Herrera & Lozano, 1996). For example, switching from one-point mutation to N-point mutation may result in more exploration during an evolutionary process, and switching from linear selection to non-linear one may change the influence weight of certain portion of individuals.

For (1), it has been discussed in the previous subsection. This subsection first reproduces DGEA falling into category (2) and then proposes how PPCea expresses experiments in category (3).

DGEA introduces a new diversity metric that computes the distance of all individuals to the average point of an N-dimensional search space. Exploration mode is identified if the diversity metric is lower than a predefined lower bound, and exploitation mode is recognized as the metric is higher than a predefined higher bound. Selection and crossover are applied to explore search space while mutation is treated as exploitation operator. Figure 4 shows the reproduction of DGEA, where *dLow* and *dHigh* are user-defined parameters and *DistanceToAvgPt* is computed by PPCea. *changeStrategy* is a PPCea statement

that switches between operators. For example, within exploration mode, tournament selection and 1-point crossover are active and mutation is halted. Conversely, mutation is the only active operator while selection and crossover are halted by the two keywords specified within the second *changeStrategy*.

```
genetic
// initialize all needed parameters including dLow, dHigh
    init;
    while ( g < Maxgen ) do
         callGA;
         if ( DistanceToAvgPt < dLow ) then // exploration mode
             changeStrategy(TOURNAMENT_SELECTION, GA_HALT_MUTATION,
         ONE_PT_CROSSOVER)
         fi;
         if ( DistanceToAvgPt > dHigh ) then // exploitation mode
             changeStrategy(GA_HALT_SELECTION, ONE_PT_MUTATION, GA_HALT_CROSSOVER)
         fi;
         g := g + 1
    end
```

Fig. 4. DGEA reproduction using PPCea

The previous example shows that PPCea can swap between different operators when needed. Halting an operator under a specific condition is also feasible by setting *GA_HALT_SELECTION*, *GA_HALT_MUTATION*, and *GA_HALT_CROSSOVER*, among others. When PPCea interpreter identifies such keywords, the operators will not be executed until they are reactivated by next *changeStrategy* statement. More details about how these are implemented will be covered in Section 3.

As mentioned before, DGEA is within category (2). A PPCea example classified within category (3) is introduced in Figure 5. Initially, *context*, a PPCea statement, defines specific operators will be executed by the evolutionary process. Line 1 shows that linear selection, 1-point mutation, and n-point crossover are picked to perform optimization search at the beginning. Lines 5 to 10 shows that two operator pairs (*TOURNAMENT_SELECTION*, *N_PT_MUTATION*) and (*RANK_SELECTION*, *ONE_PT_MUTATION*) will be swapped every 10 generations until 95th generation. Mutation will be stopped at the last 5 generations. Because *N_PT_CROSSOVER* never appears in the pairs of *changeStrategy*, this operator will remain active during the entire evolutionary process. How PPCea interpreter executes such operator adaptation will be also covered in Section 3.

2.5 Summary

As can be seen in the previous examples, the programming fashion of PPCea facilitates introducing a number of experiments with same or different settings by writing a few lines of code. Each evolutionary process run by PPCea can also be controlled deterministically or adaptively through parameter and/or operator adaptation. For space consideration, the experimental results of the examples, acting as a proof of feasibility of PPCea, are available at (Liu, 2010). Note that the previous examples also show some EAs cannot be reproduced easily derived from lacking needed attributes, multi-populations, parameters analyzed from an evolutionary process or operators. Section 3 attempts to address such problems from the perspective of EA developers. Lastly, categories of adaptive representation and adaptive fitness are also introduced in (Herrera & Lozano, 1996). They could be also potentially addressed by PPCea. Due to time constraint, they are left as one of our future work.

```
1  context(LINEAR_SELECTION, ONE_PT_MUTATION, N_PT_CROSSOVER);
2  init;
3  while (g <= Maxgen ) do //assume t = 1 initially and Maxgen = 100
4      callGA;
5      if (( g % 10) == 0) then
6          changeStrategy(TOURNAMENT_SELECTION, N_PT_MUTATION)
       //swap to tournament selection every 10 generations starting at g = 10
7      fi;
8      if (( g % 20) == 0) then
9          changeStrategy(RANK_SELECTION, ONE_PT_MUTATION)
       //swap to rank selection every 10 generations starting at g = 20
10     fi;
11     if (g > 95) then
12         changeStrategy(GA_HALT_MUTATION)
       //swap to temporarily stop mutation between generations 95 and 100
13     fi;
14     g = g + 1
15 end;
```

Fig. 5. Operator adaptation using PPCea

3. PPCea: A Portable Pattern-driven Contrivance for EA developers

Conducting parameter control experiments is always a time consuming task due to a variety of possible parameter combinations that may affect the convergence and optimization of an evolution process in different magnitudes. One may concentrate on a limited set of parameters to "de-scope" the problem (Harik & Lobo, 1999). Even so, a sufficient number of experiments are still needed due to heuristic nature of EAs. Per Aristotle, the aforementioned problems are essential difficulties (Brooks, 1987) inherent in EAs. Conversely, code snippets for computing metrics and programming logics for adapting an evolution process on-the-fly based on such metrics still scatter and tangle with other EA source code. Such inflexibility for further extension and inevitability of faulty EA source code are accidental difficulties (Brooks, 1987) that may be solved by the approaches hiding such difficulties.

A DSL is a modeling/programming language that shields accidental difficulties by introducing a higher level abstraction. It has been proved that DSLs may facilitate productivity (up to 10 times improvement), reliability, maintenability, and portability to domain users (Mernik et al., 2005). However, DSLs that are implemented by compiler or interpreter approaches may result in extension and evolution difficulties (Gray et al., 2008). For example, if a new mutation operator is introduced to PPCea, not only new syntax and semantics need to be introduced, but exsiting source code may be also affected due to inappropriate modularization in many compiler/interpreter-based DSLs including PPCea. Moreover, as mentioned in (Harik & Lobo, 1999), Holland would have never thought of a plentiful number of parameters are presented – Parameters are good for assisting in getting insight of an evolution process or helping control the process. Yet, EA computation may become overwhelmly slow resulted from parameter explosion. In summary, an objective of this section is to remedy the obstacles dervied from the introduction, extension, or evolution of parameters and operators in PPCea.

3.1 Design of PPCea

In order to design and implement PPCea in a manageable and systematic way, DSL patterns (Mernik et al., 2005) and design patterns (Gamma et al., 1995) are followed. Table 1 summarizes the DSL patterns that PPCea applies.

Workflow	Pattern	Description
Decision	Task automation System front-end	When and why to have PPCea
Analysis	FODA (Kang et al., 1990)	Find the common and variable features of EAs
Design	Denotational Semantics Design Patterns (Gamma et al., 1995)	Formally define syntax and semantics of PPCea PPCea applies composite, visitor, strategy, decorator and singleton patterns to address introduction, extension, and evolution problems.
Implementation	Interpreter	Introduce PPCea interpreter to conduct EA experiments

Table 1. The DSL patterns applied in PPCea

Decision patterns specify when and why a new DSL is essential. In order to provide an adaptable mechanism to solve such parameter control/setting problems, task automation and system front-end decision patterns are chosen. Task automation decision pattern hides the implementation details of EAs. Without browsing and understanding lengthy source code encapsulated in EAs, users omit the complex implementation but concentrate on the parameters and operators that lead to the optimization and/or convergence of EAs. Secondly, PPCea follows system front-end decision pattern that primarily handles configurations. Time-consuming and error-prone overhead can be reduced or avoided. As for analysis, PPCea utilizes Feature-Oriented Domain Analysis (Kang et al., 1990) to perform formal domain analysis so that common and variable features of EAs can be systematically identified. With such, PPCea can be formally defined using denotational semantics (Aho et al., 2007) and designed using design patterns (Gamma et al., 1995). Lastly, interpreter pattern is utilized to implement PPCea. An overview of PPCea interpreter is shown in Figure 6.

The interpreter is constructed with the assists of JFlex (Klein, 2010) and Construction of Useful Parsers (CUP) (Hudson, 2010). JFlex is a fast scanner generator for Java whose purpose is to generate a lexer that performs tokenization process for PPCea programs. CUP is a parser generator that introduces a bottom-up parser that performs syntax analysis. Such a parser may be integrated with user-defined semantics written in Java, accompanying with options to introduce syntax trees and symbol tables. The linguistic elements include commonly-seen constructs such as if-else, loop, and assignment statements as well as expressions and operators to perform necessary parameter adjustments. Additionally, domain-specific elements to describe an EA are presented: *init* statement initializes a population, *callGA* statement performs a GA, *callES* statement performs an ES, *resize* statement allows population resizing, *changeStrategy* statement offers the potentials to switch between different operators on-the-fly, *context* statement determines the operators that constitutes an EA, and *require* statement determines which domain-specific parameters to be computed. Such parameters are either the results from an evolutionary process that can be also acted as metrics or feedbacks to assist parameter control. There are also miscellaneous statements for various purposes (e.g., *IOStatement*). Interested readers may find more information on the PPCea web page (Liu, 2010). All the above linguistic elements

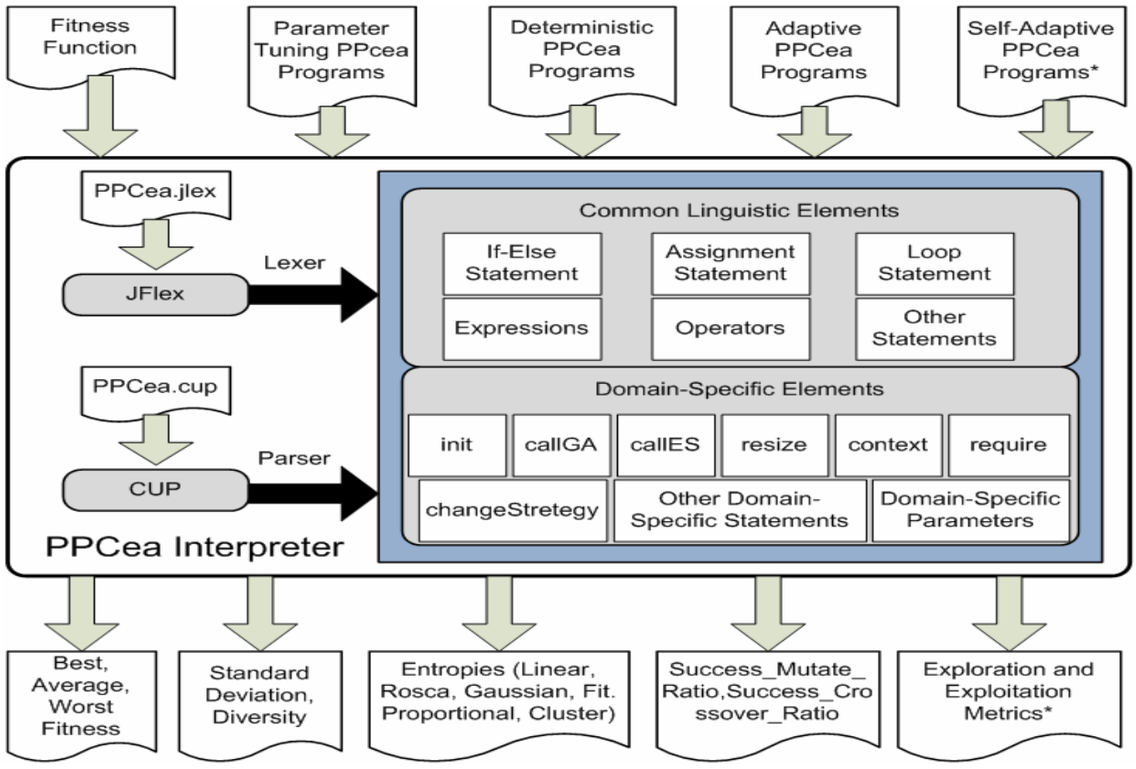


Fig. 6. An overview of PPCea interpreter (* means ongoing/future tasks)

are represented as Java classes embedded with associated semantics. The interpreter currently accepts parameter tuning, deterministic, and adaptive PPCea programs as inputs, as seen at the top of the figure. The outputs, at the bottom of the figure generated by the interpreter, comprise best, average, worst fitness, standard deviation and Euclidean distance (i.e., diversity), entropy (Liu et al., 2009), and the success rates of crossover and mutation, among others. More domain-specific constructs and parameters can be introduced, extended and evolved following the design patterns introduced in the subsequent subsections.

3.2 Evolutionary Algorithm and operator introductions

Since Evolutionary Algorithms (EAs) were coined, there have been a variety of algorithms, operators, and parameters proposed in order to apply to a various number of applications and experiments as well as further improve optimal results and/or convergence rate of such algorithms. For example, different from the general sketches of GAs in (Michalewicz, 1996), Bi-population GA (Tsutsui et al., 1997), aGA (Ghosh et al., 1996), and PROFIGA (Eiben et al., 2004), among others are variations that respectively introduces new algorithmic strategies (e.g., splitting populations into exploration and exploitation modes), new attributes (e.g., ages for individuals) or new operators (population resizing) to facilitate optimization and/or convergence. Additionally, DGEA, Evolution Strategies using Cauchy Distribution (Yao et al., 1999), Particle Swamp Optimization (Kennedy and Eberhart, 2001), and Differential Evolution (Storn & Price, 1997), to name a few, are also categorized in EAs that solve optimization problems from other perspectives. In addition to algorithm

introductions, many variations of existing operators and domain-specific parameters are also introduced (e.g., tournament selection, linear selection, uniform crossover, intermediate crossover, diversity-to-average measure (Ursem, 2002), and cluster entropy (Liu et al., 2009)). PPCea has anticipated such extension and evolution potentials and hence adopted design patterns so that future changes can be addressed with ease.

3.2.1 Evolutionary Algorithm and operator introductions using composite pattern

Because PPCea is developed by following the interpreter/compiler pattern (Mernik et al., 2005), inevitably, the syntactical representation of a PPCea program is expressed as a syntax tree structure (Aho et al., 2007). However, a commonly-seen implementation issue existing in such a tree structure is to deal with the composite-atomic hierarchies (i.e., whole-part hierarchies). For example, PPCea comprises if-else and loop statements that may embrace zero or more composite and/or atomic statements as child nodes (e.g., a nested if-else statement); and conversely, assignment and domain-specific statements are atomic ones that cannot hold any statement nested within their bodies. Because composite statements are derived from recursive productions defined in PPCea grammar (see appendix), they do not posses concrete semantics as other statements do. To reduce implementation complexity, a synergistic objective needs to be fulfilled: How to uniformly treat composite and atomic language constructs in the tree structure (i.e., hide the differences), while distinctions between these two types of language constructs can still be easily made if necessary (i.e., behave as atomic and composite ones as supposed).

A primary objective of composite pattern (Gamma et al., 1995) is to represent whole-part hierarchies and achieve the synergistic objective mentioned above. Figure 7 shows the implementation of composite pattern applied to PPCea interpreter, where *IStmt* is an abstract class that defines the interface and common behavior of both atomic (i.e., *Stmt*) and composite (i.e., *Series*) statements. The advantages of composite pattern mainly lie in the introduction of *IStmt* and the composition between *Series* and zero to more *IStmt* objects, which will be later identified as *Series* or *Stmt* concrete objects using polymorphism.

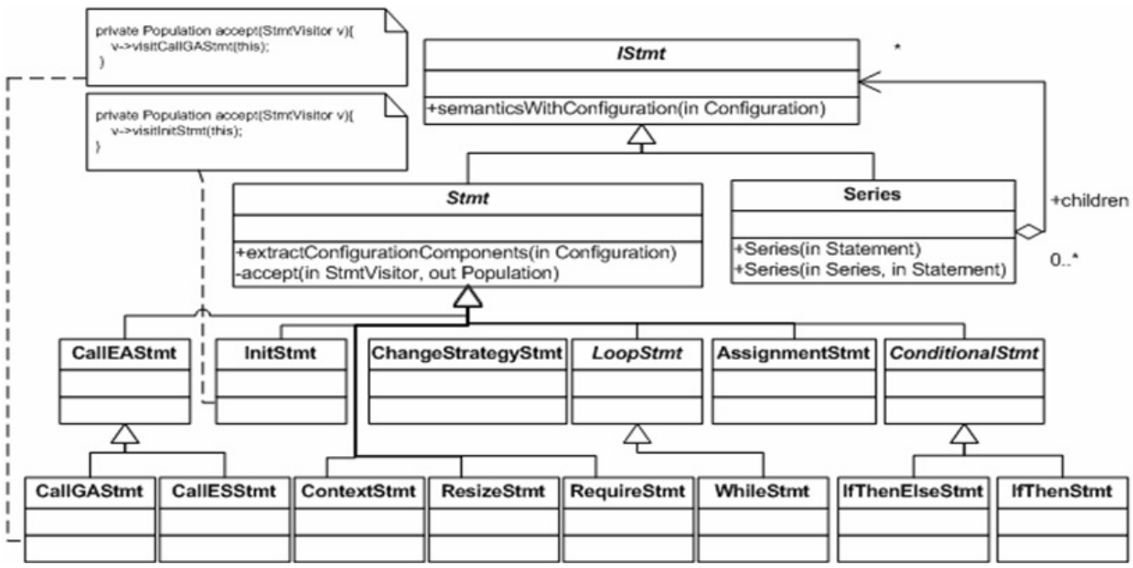


Fig. 7. Composite pattern applied to PPCea intepreter

Composite pattern also follows open-close principle (Meyer, 2000). Such a principle advocates "open for extension and close for modification": New statements can be added through inheritance; and modification on interface is closed and modification on implementation is isolated to associated methods only. For example, if one requests to introduce *callPSO* statement for Particle Swamp Optimization (Kennedy & Eberhart, 2001) in PPCea, three steps will be needed at the lexical, syntactical and semantic levels: (1) *callPSO* needs to be introduced as a token in PPCea.jlex (as seen in Figure 6); (2) A terminal that represents *callPSO* statement and the associated syntax are requested in PPCea.cup (can be found in Figure 6 too); and (3) Introduce a *CallPSOStmt* class inherited from *CallEAStmt*. Such a class defines the semantics/algorithms of Particle Swamp Optimization. A new operator that does not relate to any specific algorithm may be also introduced in the same manner. With such, introducing new algorithms or operators will not interfere with the remaining parts of PPCea. Extension and evolution of evolutionary algorithms and operators will be introduced next. Introducing evolutionary operators and parameters comprises the same steps as mentioned. For example, because *ResizeStmt* and *ChangeStrategyStmt* are two operators that can be applied to various EAs, they are not encapsulated into specific EA statements. For EA-specific operators, from the implementation's perspective, they can be introduced as standalone statements like r*esize* and *changeStrategy* or they can be encapsulated as methods in associated EA statement classes. For the sake of better design to satisfy high cohesion and responsibility driven concepts (Schach, 2010), such operators are encapsulated into EA associated statements.

3.3 Evolutionary Algorithm and operator extensions/evolutions using visitor pattern

Although composite pattern achieves the synergistic objective that allows uniformed treatments and making distinctions on atomic and composite statements when needed, extending or evolving methods encapsulated in EA statements is difficult. (Ironically, they are resulted from following good design principles as mentioned in the previous section). For example, *semanticWithConguration* in *CallEAStmt* is derived from *IStmt* that defines the semantics of a statement by executing a set of evolutionary operators (e.g., mutation, crossover, and selection). If a new operator, e.g., elite or n-point mutation, is introduced in *CallEAStmt* and invoked by *semanticWithConguration*, all *CallEAStmt*'s subclasses will be affected and recompilation is requested. Additionally, any evolution change to the existing algorithms and operators may be scattered around the entire class, which could be error-prone and resulted in regression faults. Because visitor pattern (Gamma et al., 1995) has succeeded in solving such tree-related problems with composite pattern (e.g., Wu et al., 2005), PPCea adopts visitor pattern so that the aforementioned extension and evolution problem can be solved.

As shown in Figure 8, PPCea introduces a super class, called *StmtVisitor*, which comprises two subclasses: *EAStmtVisitor* and *GenericStmtVisitor*, where the former one is to define the semantics of EA-specific operators and the latter one is to define the semantics of generic or non-EA-specific operators. For EA-specific operators, three important evolutionary operators (*selection*, *crossover*, and *mutation*) and two utility operators (*eval* and *getStat*) are introduced as *EAStmtVisitor*'s subclasses, where *eval* is to compute the fitness value of each individual and *getStat* is to compute the statistical data of an EA that are shown at the bottom of Figure 6. For generic operators, population initialization (*init*), population resizing

(*resize*), strategy changing (*changeStrategy*) for adapting different evolutionary operators on-the-fly, are introduced. Note that *Context* class introduced in Figure 8 is initialized by *ContextStmt* in Figure 7 that will store the current statement (or EA operator) that PPCea interpreter is executing and the resultant population. The usage of this class will be explained in more details in the next subsection.

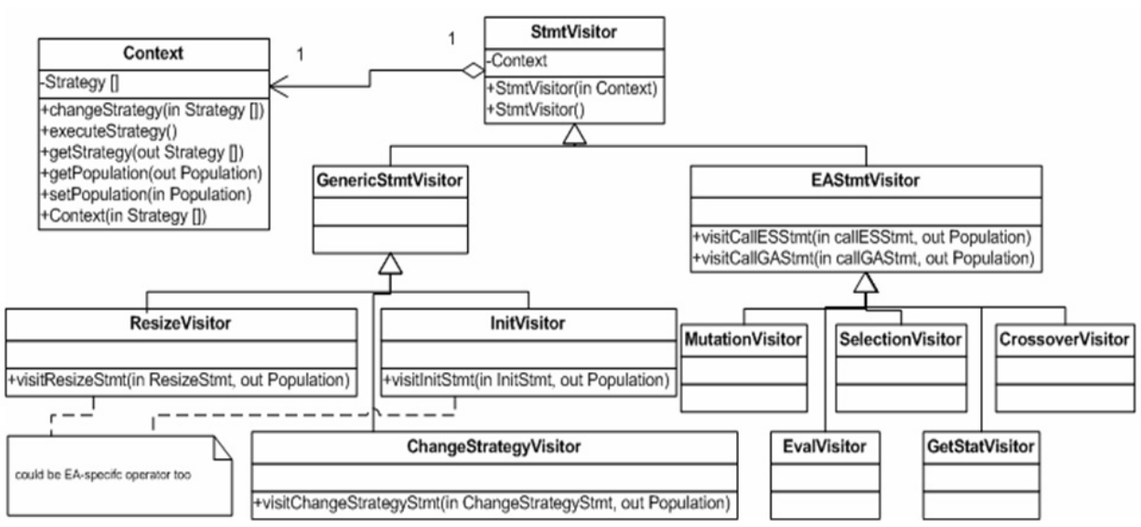


Fig. 8. Visitor pattern applied to PPCea

Readers who are not familiar with design patterns may find it difficult to see how composite and visitor patterns work together to tackle introduction, extension, and evolution problems by decoupling the syntax tree structure and operators within each statement, while at the same time allow the semantics of each statement as well as the entire program to be functioned correctly. To explain such correlation, the first step is to understand how CUP works with associated semantics written in Java classes. Because CUP is a compiler generator that generates a bottom-up parser, when each non-terminal is traversed, the corresponding Java class is instantiated. All the necessary classes (statements, expressions, and operators as seen in Figure 6) that define associated semantics will be available after the root of the parse tree is traversed. Then the root node will trigger semantics of each line of program to be interpreted and executed. When a statement, called *init*, is reached, the *semanticWithConfiguration* method within such a class will invoke a private method, called *accept* with an object of *InitVisitor* class passed in (see Figure 7). Within the *accept* method, *visitInitStmt* method of *InitVisitor* class will perform node/statement identification, called double dispatch (Gamma et al., 1995). If the current statement to be executed is *init*, *visitInitStmt* will execute the semantics of population initialization accordingly. Similarly, if *callGA* (or *callES*) is executed, objects of EvalVisitor, SelectionVisitor, MutationVisitor, CrossoverVisitor, and GetStatVisitor will be passed as parameters of *accept* method. The semantics defined in each visitor subclass will be executed after *callGA* statement is identified by double dispatch.

Important advantages that visitor pattern is capable of attacking extension and evolution problem can be expressed as the following three examples:

1. If a new EA-related operator is requested (e.g., *clone* or *repair*), we do not introduce an operator in *CallEAStmt*, *CallGAStmt* or *CallESStmt*, which will result in recompilation of almost the entire composite class hierarchy of Figure 7, as mentioned in the previous subsection. Instead, a subclass of *EAStmtVisitor* that defines the semantics of the newly introduced operator can be extended/introduced without affecting the remaining part of PPCea. There is no need to revise any part of JFlex and CUP files of PPCea either;
2. If a new generic operator is needed (e.g., *randomize* that introduces new random individuals into current population), a subclass that defines such an operator can be inherited from *GenericStmtVisitor* without editing other parts. If the new operator is also requested to be added to PPCea grammar, there is a need to introduce an associated token, syntax, and a subclass of *Stmt* respectively at the lexical, syntactical and semantic levels. Note, however, such extensions will still not interfere other parts of the existing code; and
3. If an existing EA-specific or generic operator is requested to be changed (i.e., evolution), the focus will be only on the specific subclass. Other parts will not be emphasized so the opportunities of regression faults will be minimized.

Although utilizing composite and visitor patterns to solve tree structure problems is not new, our implementation slightly varies the traditional solution and results in an additional advantage that can be observed in Figure 8: Even though introducing an EA at the statement level (e.g., *callGA*) may give readers impression that it is inflexible to control lower level evolutionary operators. Instead, it is a wrong impression! The visitor pattern utilized in PPCea is a variant – it is implemented along with strategy pattern (Gamma et al., 1995), where different evolutionary operators can be controlled through *changeStrategy* at the granularity of operator rather than algorithm. Such implementation avoids possible frequent recompilation while allowing ease of extension and evolution. Namely, only when a new EA, for example, *callPSO,* is introduced, existing subclasses of *EAStmtVisitor* need to add and compile a new operator, called *visitCallPSO.* More discussions will be covered in Section 3.5.

3.4 Strategic operator adaptation

Section 2.4 introduced three categories of operator adaptation: (1) Adaptation delegated to parameters (i.e., parameter control); (2) Adaptation among different types of operators; and (3) Adaptation among same types of operators. Strategy pattern (Gamma et al., 1995) is applied and integrated with visitor pattern to realize categories (2) and (3).

A primary objective of strategy pattern is to introduce a set of functionalities that can be interchanged upon request. Different kinds of evolutionary operators for GAs and ESs are introduced as subclasses of *Strategy* class in Figures 9 and 10, respectively. For example, we have implemented linear, non-linear, ranking, tournament and roulette wheel selections as subclasses of *GASelectionStrategy*. Similarly, the implementation of one-point and n-point mutation/crossover is defined in the associated subclasses of *GAMutationStrategy*/*GACrossoverStrategy*. If there is more than one way to initialize population, subclasses that specify such differences can be inherited from *GAInitStrategy*.

Extension and evolution to different algorithms of an operator can be also easily done by introducing subclasses. For example, fitness proportional selection may be introduced as a subclass of *GASelectionStrategy* without interfering the remaining parts of the source code;

and revising existing strategies will be isolated in their own classes – Again, this design follows open-close principle.

Figure 5 in Section 2.4 has presented how to adapt operators using PPCea. The dynamics of such code snippet with respect to strategy pattern is summarized as follows. Figure 10 is a simplified version of Figure 5 for ease of reading. Line 1 sets the default operators for selection, mutation, and crossover into context. A *Context* object will be instantiated with the *Strategy* objects of linear selection, 1-point mutation and n-point crossover as parameters.

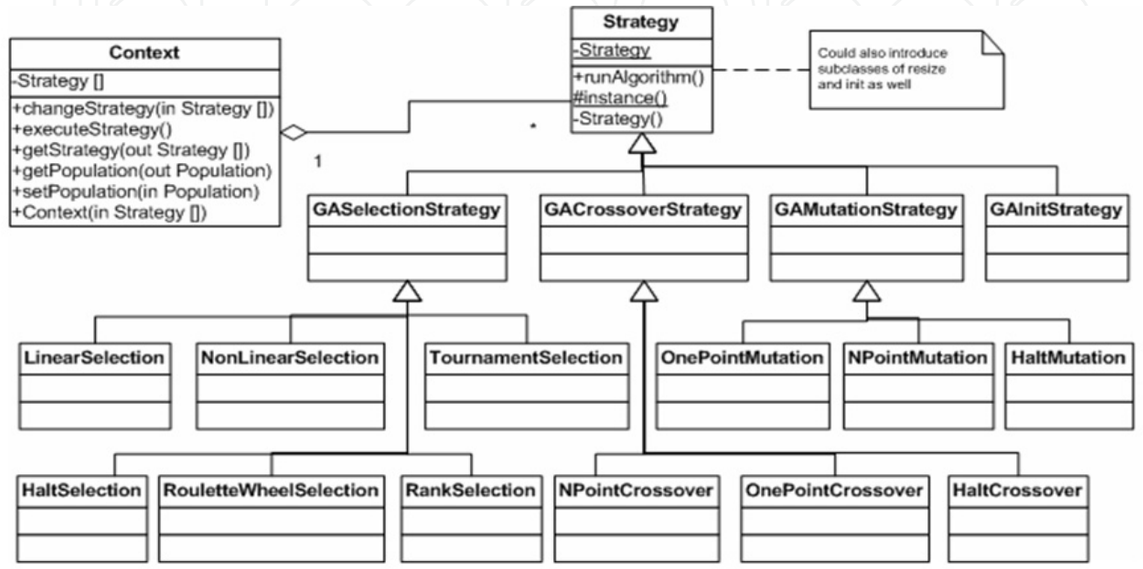


Fig. 9. Strategy pattern applied to GAs in PPCea

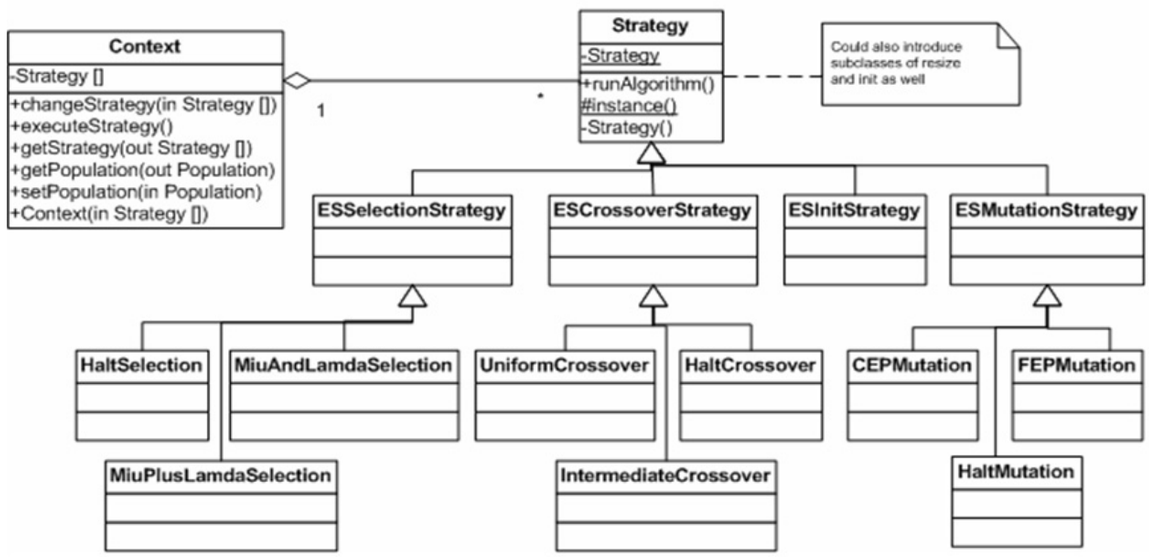


Fig. 10. Strategy pattern applied to ESs in PPCea

Line 2 initializes a population for the GA. Then *callGA* performs the GA using such strategies. Namely, *CallGA* statement in Figure 7 will invoke its *semanticWithConfiguration* method that accepts objects of *SelectionVisitor*, *MutationVisitor*, *CrossoverVisitor*, *EvalVisitor*, and *GetStatVisitor* shown in Figure 8. Each visitor object accesses the *Context* object and executes the associated *Strategy* object within *visitCallGAStmt* method. For example, the object of *SelectionVisitor* by default will execute an object of *LinearSelection* set in Line 1 of Figure 11. Between two *if*-statements, by using *changeStrategy* selection operator switches between tournament and rank selections every 10 generations. Additionally, mutation also switches between 1-point and n-point mutations every 10 generations. Crossover, on the other hand, will remain during the entire evolutionary process. Such behaviour is performed based on the following steps: (1) *ChangeStrategy* statement accepts an object of *ChangeStrategyVisitor*, which allows new *Strategy* objects (e.g., rank selection and n-point mutation at generation 10) to interchange with the existing *Strategy* object (e.g., linear selection and 1-point mutation). After exchange, *Context* object will execute new strategies. Note that, to avoid class explosion, Singleton pattern (Gamma et al., 1995) is applied to force each operator only has one associated object instantiated all the time. So that when the strategy pairs (tournament selection and n-point mutation) and (rank selection and 1-point mutation) are swapped every 10 generations, there is no new object instantiated, but the existing ones are reused. Halting an operator temporarily is also a feasible solution by replacing the current *Strategy* object to *HaltSelection*, *HaltMutation*, or *HaltCrossover*, which simply choose not to execute the current strategies. With such, EAs categorized as uni-process approaches in (Liu et al., 2009) can be reproduced using PPCea (e.g., DGEA in Figure 4).

```
1 context(LINEAR_SELECTION, ONE_PT_MUTATION, N_PT_CROSSOVER);
  //... skip some code
2 callGA;
3 if (( t % 10) == 0) then
4     changeStrategy(TOURNAMENT_SELECTION, N_PT_MUTATION) then
5 fi;
6 if (( t % 20) == 0) then
7     changeStrategy(RANK_SELECTION, ONE_PT_MUTATION) then
8 fi;
  //...skip some code
```

Fig. 11. A simplified version of Figure 5.

3.5 Parameter extension and evolution

Domain-specific parameters are those predefined in a DSL grammar and may facilitate productivity and other advantages of DSLs mentioned before. In PPCea, domain-specific parameters, shown in Table 2, can be categorized in two groups: (1) Parameters that are used for controlling an evolutionary process; and (2) Parameters that are computed at the end of each generation and may be treated as feedback to adjust an evolutionary process.

As seen in Table 2, parameters in group (1) are quite diverse. Some may be accessed across the entire project (e.g., *Popsize*, *Maxgen*, and *Epoch*) and others may be used by specific operators (e.g., *Alpha*, *Beta*, *Miu*, and *Lamda*). From the perspective of compiler/interpreter implementation, this kind of parameters usually already has identities stored in a symbol table (i.e., predefined). When such parameters are initialized by assignment statement, their values are stored in the symbol table accordingly. Whenever and wherever needed, the

values can be accessed through the symbol table. Extension of parameters falling in group (1) usually means introducing new domain-specific notations at the lexical, syntactical and semantic levels, which therefore has the same way of implementing domain-specific statements. Conversely, evolution for such a kind of parameters is usually renaming and

Group	Parameter Name	Description
(1)	Function	Fitness function to be evaluated (Obtained from (Yao et al., 1999))
	Popsize	Number of individuals of a population
	Maxgen	Maximum number of generation for an evolutionary process
	Epoch	Generation stride for parameter control adaptation
	pm	Mutation rate
	pc	Crossover rate
	psr	Stochastic ranking rate
	Alpha	Selection pressure (α) for linear/nonlinear selection
	Beta	Selection pressure (β) for linear/nonlinear selection
	Miu	Selection parameter (μ) for ESs
	Lambda	Selection parameter (λ) for ESs
	TourQ	Selection parameter for tournament selection
	KMeans	Number of centroids for clustering entropy
(2)	Best	Best fitness value of all individuals
	Average	Average fitness value of all individuals
	Worst	Worst fitness value of all individuals
	RatioM	Success mutation rate
	RatioC	Success crossover rate
	Stdv	Standard deviation of all individuals
	Euclidean	Euclidean distance of all individuals
	LinearEntropy	Linear Entropy (Liu et al., 2009)
	RoscaEntropy	Rosca Entropy (Liu et al., 2009)
	GaussianEntropy	Gaussian Entropy (Liu et al., 2009)
	FitProEntropy	Fitness Proportional Entropy (Liu et al., 2009)
	ClusterEntropy	Clustering Entropy (Liu et al., 2009)

Table 2. Current domain-specific parameters introduced in PPCea

changing valid scope based on our experience. Hence, refactoring (Fowler, 1999) may be applied to the lexical, syntactical and semantic levels to tackle evolution. Parameters in group (2) are computational results analyzed from population either after every (several) generation(s) or at the end of an entire evolutionary process. However, not all of such parameters are needed all the time. For example, for parameter tuning approaches, readers may be interested in fitness-related parameters only. Similarly, for non-entropy-driven approaches, one may avoid the computation of the five entropies shown in Table 2 and hence improve the performance. To achieve such a "pay-as-you-go" objective, decorator pattern is applied. As seen in Figure 12, *Parameter* class is introduced as a super class that applies singleton pattern to avoid more than one instance instantiated during an evolutionary process. *Decoratee* class is a "decoratee" super class, which means that all the objects of the subclasses inherited from *Decoratee* are mandatory to be provided by PPCea interpreter. Conversely, *DecoratorParameter* represents a super class whose subclass objects can be optionally computed upon requests.

The dynamics of how "pay-as-you-go" is achieved may be further observed by the code snippet shown in Figure 13: *visitCallGAStmt* is a method defined within *GetStatVisitor* class, accepted by *CallGAStmt* as described before, whose purpose is to analyze statistical results of a population. Lines 3 to 5 specify the mandatory domain-specific parameters to be computed. From lines 6 to 20, users may determine if any object of *Decorator* requires computation or not either directly defined in PPCea code (e.g., `require(LINEAR_ENTROPY, FITPRO_ENTROPY, ROSCA_ENTROPY);`) or through the graphical user interface we developed for PPCea interpreter. For example, if linear, fitness proportional and Rosca entropies are

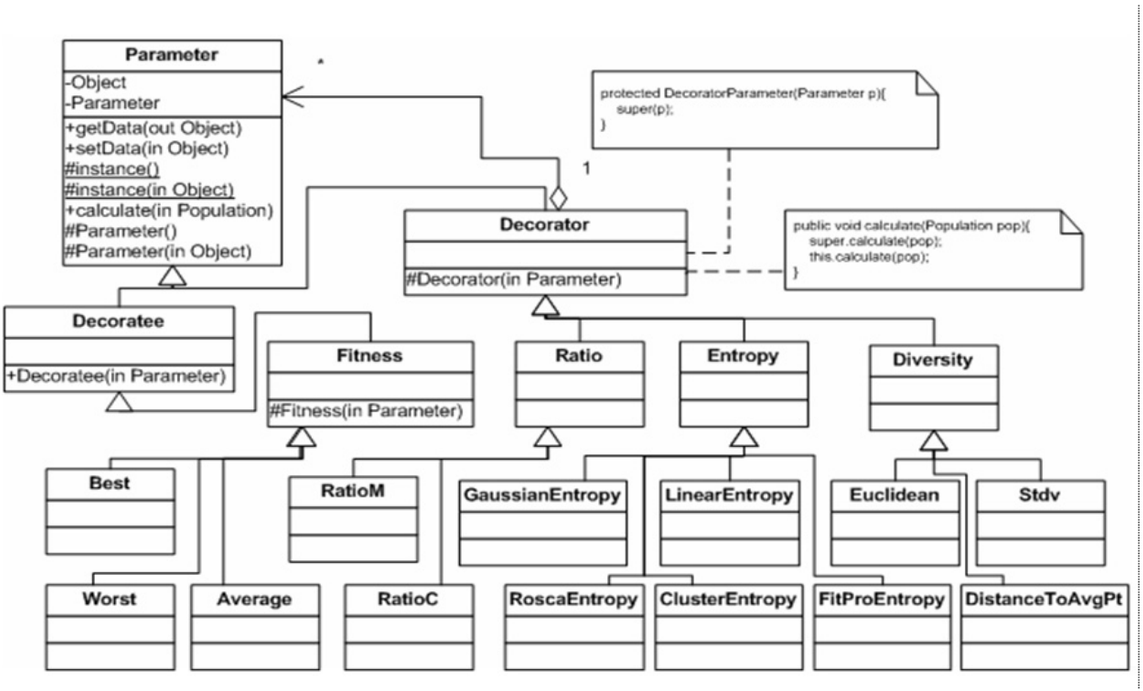


Fig. 12. Decorator pattern applied to PPCea.

selected by users (as seen above), when line 21 of Figure 13 is invoked, the *calculate* methods in *Best*, *Worst*, *Average*, *LinearEntropy*, *FitProEntropy*, and *RoscaEntropy* will be executed in a cascading and sequential order (see the notes in Figure 12).

How does decorator pattern address the extension and evolution problems of parameters in group (2)? For extension, if a new parameter is mandatory, a subclass should be inherited from *Decoratee* class. Conversely, if users are in charge of the computation necessities of newly introduced parameters, subclasses of *Decorator* will be introduced. The computational algorithms of the new parameters will be defined in their own *calculate* methods, along with an invocation to its super class' *calculate* method. One drawback of applying Decorator pattern is that when extension (i.e., introducing a new domain-specific parameter of group (2)) occurs, inevitably Figure 13 needs to be revised to incorporate such a change. It is because the purpose of the *if*-statements is to retrieve the answer of optional parameters that users determine to include. As for evolution, if any computational algorithm of a parameter changes, it is isolated in the associated *calculate* method.

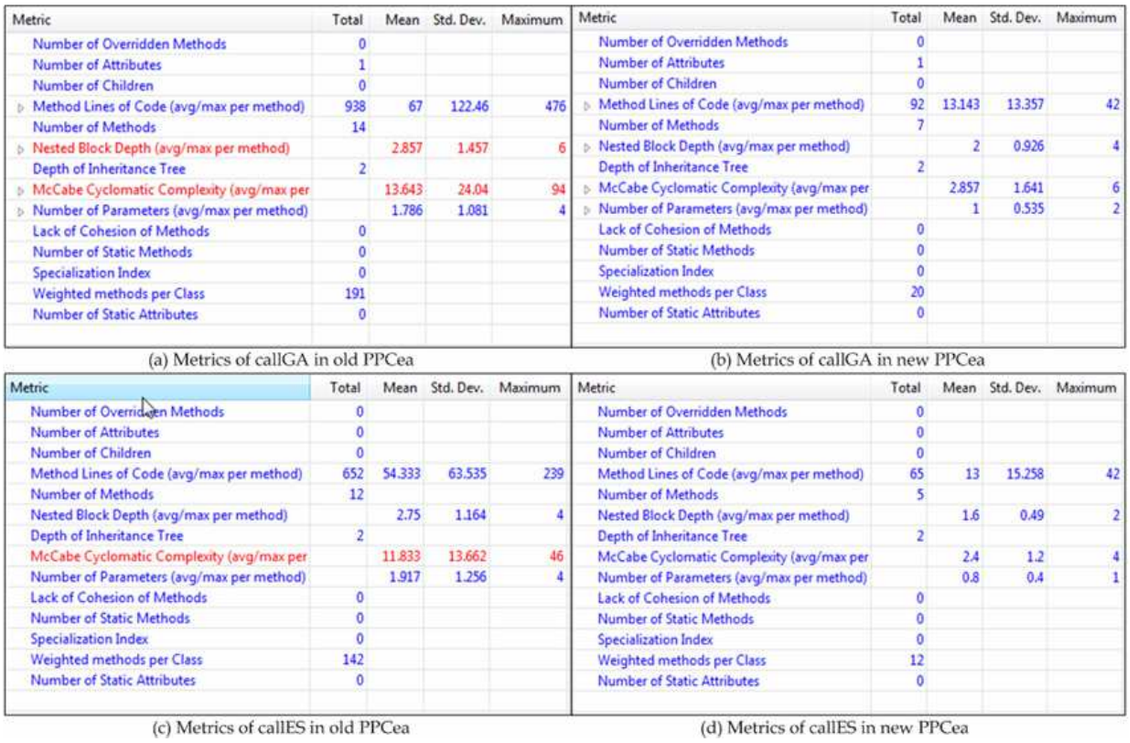
```
1 Individual[] visitCallGAStmt(CallGAStmt gaStmt){
2   /* skip the code of retrieving parameters from symbol table ...*/
3   Parameter decorator = new Best(); //required parameter
4   decorator = new Worst(decorator); //required parameter
5   decorator = new Average (decorator); //required parameter
6   if (Linear_Entropy == 1) {
7        decorator = new LinearEntropy(decorator); //optional parameter
8   }
9   if (Gaussian_Entropy == 1) {
10       decorator = new GaussianEntropy(decorator); //optional parameter
11  }
12  if (FitPro_Entropy == 1) {
13       decorator = new FitProEntropy(decorator); //optional parameter
14  }
15  if (Rosca_Entropy == 1) {
16       decorator = new RoscaEntropy(decorator); //optional parameter
17  }
18  if (Cluster_Entropy == 1) {
19       decorator = new ClusterEntropy(decorator); //optional parameter
20  }
21  decorator.calculate(population);
22  return population;
23 }
```

Fig. 13. Decorator pattern applied to PPCea.

3.6 Software metrics

Software metrics (Lincke et al., 2008) are measures that assist in providing comprehensibility of software being assessed. Therefore, the quality of the software can be observed through such metrics. PPCea utilizes Eclipse Metrics plug-in (Sauer, 2010) to compare the current version implemented with DSL and design patterns and the original version introduced in (Liu et al., 2004), shown in Figure 14.

As seen in the figure, PPCea using patterns has much higher design and implementation quality in terms of Method lines of code, McCabe cyclomatic complexity (suggested maximum value: 10), and weighted methods per class. All other metrics listed in Figure 14 surpass the suggested maximum values. Namely, the design and implementation of PPCea with patterns is in good quality and refactoring may not be necessary.



Metric	Total	Mean	Std. Dev.	Maximum
Number of Overridden Methods	0			
Number of Attributes	1			
Number of Children	0			
Method Lines of Code (avg/max per method)	938	67	122.46	476
Number of Methods	14			
Nested Block Depth (avg/max per method)		2.857	1.457	6
Depth of Inheritance Tree	2			
McCabe Cyclomatic Complexity (avg/max per		13.643	24.04	94
Number of Parameters (avg/max per method)		1.786	1.081	4
Lack of Cohesion of Methods	0			
Number of Static Methods	0			
Specialization Index	0			
Weighted methods per Class	191			
Number of Static Attributes	0			

(a) Metrics of callGA in old PPCea

Metric	Total	Mean	Std. Dev.	Maximum
Number of Overridden Methods	0			
Number of Attributes	1			
Number of Children	0			
Method Lines of Code (avg/max per method)	92	13.143	13.357	42
Number of Methods	7			
Nested Block Depth (avg/max per method)		2	0.926	4
Depth of Inheritance Tree	2			
McCabe Cyclomatic Complexity (avg/max per		2.857	1.641	6
Number of Parameters (avg/max per method)		1	0.535	2
Lack of Cohesion of Methods	0			
Number of Static Methods	0			
Specialization Index	0			
Weighted methods per Class	20			
Number of Static Attributes	0			

(b) Metrics of callGA in new PPCea

Metric	Total	Mean	Std. Dev.	Maximum
Number of Overridden Methods	0			
Number of Attributes	0			
Number of Children	0			
Method Lines of Code (avg/max per method)	652	54.333	63.535	239
Number of Methods	12			
Nested Block Depth (avg/max per method)		2.75	1.164	4
Depth of Inheritance Tree	2			
McCabe Cyclomatic Complexity (avg/max per		11.833	13.662	46
Number of Parameters (avg/max per method)		1.917	1.256	4
Lack of Cohesion of Methods	0			
Number of Static Methods	0			
Specialization Index	0			
Weighted methods per Class	142			
Number of Static Attributes	0			

(c) Metrics of callES in old PPCea

Metric	Total	Mean	Std. Dev.	Maximum
Number of Overridden Methods	0			
Number of Attributes	0			
Number of Children	0			
Method Lines of Code (avg/max per method)	65	13	15.258	42
Number of Methods	5			
Nested Block Depth (avg/max per method)		1.6	0.49	2
Depth of Inheritance Tree	2			
McCabe Cyclomatic Complexity (avg/max per		2.4	1.2	4
Number of Parameters (avg/max per method)		0.8	0.4	1
Lack of Cohesion of Methods	0			
Number of Static Methods	0			
Specialization Index	0			
Weighted methods per Class	12			
Number of Static Attributes	0			

(d) Metrics of callES in new PPCea

Fig. 14. Metrics comparison between old and new PPCea

3.7 Potentials of PPCea

The first five subsections discuss how to overcome introduction, extension and evolution problems in four specific levels: (1) Evolutionary algorithms; (2) Evolutionary and generic operators; (3) Functionalities (i.e., strategies) of an operator; and (3) Domain-specific parameters. For (1), composite pattern facilitates the introduction/extension of evolutionary algorithms by introducing EAs as domain-specific statements, subclasses inherited from *IStmt*. For (2), visitor pattern promotes introduction/extension of evolutionary and generic operators by introducing subclasses of *StmtVisitor*. If evolutionary algorithms or its operators evolve, the changes will be isolated in the subclasses of *StmtVisitor* (or associated strategies), because such subclasses define the semantics of PPCea statements. Additionally, the decision of not introducing EA-specific operators at the PPCea statement level reduces the possibility of frequent changes/recompilation of the *StmtVisitor* and its subclasses. For (3), strategy pattern assists introduction/extension of different algorithmic strategies of an operator by introducing subclasses of *Strategy*. Evolution of such strategies is also isolated in associated classes. Also, PPCea is capable of adapting with operators on-the-fly under the support of strategy pattern. Lastly, decorator pattern addresses the problem of introduction/extension and evolution of domain-specific parameters by introducing subclasses of *Decorator* and *Decoratee*. Users are also allowed to determine which parameters to be analyzed so that unnecessary computation cost can be reduced.

To illustrate how a new EA can be introduced or an existing EA can be reproduced, let us use GAVaPS as an example. The algorithm of GAVaPS is as follows.

```
begin
  t=O
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
    begin
    t = t + l
    increase the age of each individual by 1
    recombine P(t)
    evaluate P(t)
    remove from P(t) all individuals with age greater than the lifetime
  end
end
```

Fig. 15. The GAVaPS algorithm from (Arabas et al., 1994)

Based on (Arabas et al., 1994), *recombine* in Figure 15 performs normal mutation and crossover and then selection chooses offspring from all individuals with equal opportunity. As for *remove*, it will kill all individuals older than a predefined *lifetime* threshold. The population will be then resized based on the formula defined in the paper. To realize GAVaPS using PPCea, age attribute needs to be introduced in *Individual* class. Also, *lifetime* parameter may be introduced as a parameter in group (1). With such, users can adjust the value of *lifetime* by PPCea code. Then *GAVaPSSelection*, inherited from *GASelectionStrategy* in Figure 9, implements selection mechanism with equal opportunity and increments age if needed. Because there is more than one *resize* algorithm, a *ResizeStrategy* subclass may be inherited from *Strategy*. Then *ResizeByAge* may be introduced as a subclass of *ResizeStrategy* that kills all overage individuals and randomly introduces the number of new individuals using the formula if needed. Figure 16 is a pseudo PPCea code to simulate Figure 15 under the assumption all necessary subclasses are introduced in PPCea. Another possible implementation option is to introduce an entire new PPCea statement, called *callGAVaPS.* Nothing is really different except that *callGAVaPS* encapsulates all needed *Visitor* objects, which invoke objects of *GAVaPSSelection, OnePointMutation, OnePointCrossover,* and *ResizeByAge.*

```
Lifetime := 5; // any individual older than 5 will be killed
context(GAVAPS_SELECTION, ONE_PT_MUTATION, ONE_PT_CROSSOVER,
RESIZE_BY_AGE);
init;
while ( g < Maxgen ) do
  callGA;
  Popsize := ... // ... means the formulae from Arabas et al. 94
  resize( Popsize );
  g := g + 1
end
```

Fig. 16. The pseudo PPCea code that reproduces GAVaPS

In summary, PPCea utilizes DSL patterns and five design patterns so that the introduction/extension and evolution problems at the algorithm, operator, strategy, and parameter levels can be respectively addressed.

4. Related work

Evolving Objects (Keijzer et al., 2002) is an evolutionary computation framework that constructs an EA through component composition. Namely, each EA operator/statement is considered as a component and users need to select which specific components (similar to strategies) to be filled in to a specific spot of an evolutionary process. User defined parameters can be introduced to a file, which will be interpreted by the framework. ECJ (Luke et al., 2010) is a Java-based evolutionary computation system that requests users to describe an EA in Java by reusing/invoking a great number of packages for different EA operators. A set of predefined parameters with fixed identities also need to be defined in a specific file, acting like domain-specific parameters in PPCea. ESDL (Dower & Woodward, 2010) is a DSL that introduces SQL-like syntax for users to construct EAs. Name conventions for both EA operators and parameters need to be followed, which is same as PPCea. Because of interoperability advantage of XML, Veenhuis et al. introduced EAML (Veenhuis et al., 2000), a modeling language that utilizes XML to represent an EA. With such, different EA framework/software may introduce their own evolutionary processes by interpreting EAML files. There are also many EA framework or software that the book chapter is not able to fully cover. We leave this part to interested readers.

5. Conclusion

Controlling parameter settings to reach optimization and/or convergence of an EA has been a challenging topic in the evolutionary computation community. Firstly, due to meat-heuristic and stochastic nature, there is a need to conduct a sufficient number of experiments of an EA under different parameter settings. Additionally, many practitioners and scholars have put forth various algorithms, operators, and parameters to improve the optimization and/or convergence. Without automatic tools for EA users, conducting EA experiments would become tedious and error-prone. Without capabilities to extend and evolve automatic tools, EA developers would not be able to invent new algorithms, operators, strategies, and parameters. PPCea offers a synergistic solution to address the aforementioned problems from the perspectives of both users and developers. The contributions of PPCea are three-folds: (1) PPCea is an automatic EA tool in a language format that assists EA users to conduct experiments using three parameter setting approaches introduced by Eiben et al.; (2) PPCea is an open-ended EA tool that allows EA developers to introduce, extend and evolve EA constructs in algorithm, operator, strategy and parameter levels; and (3) PPCea offers a fair platform to perform EA comparison – both reproduced algorithms and new algorithms can be described in PPCea code and run under PPCea interpreter.

We have identified several future directions: (1) Multi-populations and multi-objective EAs are missing in current version. With such, more EAs can be reproduced (e.g., parameter-less GA) and invented; (2) As can be seen in Figure 6, PPCea currently cannot handle self adaptation algorithms. How to represent such algorithms and still offer open-end solutions is an emerging issue to tackle; (3) With the metrics from (Črepinšek et al., in press) introduced to PPCea, explicit balance between evolutionary and exploitation in a programmable fashion can be foreseen; and (4) Existing algorithms, operators, and strategies are effective in the individual granularity. Similar algorithms, operators, and

strategies working at the genotypical level may result in finer-grained experiments. With the aforementioned issues resolved, more EA users and developers may be benefited by PPCea.

Appendix: PPCea Grammar

```
Program          -> genetic Series end genetic
Series           -> Series Statement ; Statement
Statement        -> if Cond then Series fi | if Cond then Series else Series fi |
                    while Cond do Series end | ID := E |
                    init | callGA | callES | resize( N ) |
                    context ( Strategies ) | changeStrategy( Strategies ) |
                    require ( Parameters ) |
                    read Ids | write Ids |
                    readfile FileName | writeresult |
                    validateMsg | invalidateMsg

Condition        -> (Expression RelationOp Expression)
Expression       -> Expression Operator Expression | (Expression) |
                    sqrt(Expression) | exp(Expression) | Number | Double | ID |
Operator         -> + | - | * | / | && | ||
RelationOp       -> < | > | <= | >= | != | ==
Strategies       -> Strategies , Strategy
Strategy         -> Selection | Mutation | Crossover
Selection        -> LINEAR_SELECTION | NON_LINEAR_SELECTION |
                    TOURNAMENT_SELECTION | RANK_SELECTION |
                    ROULETTEWHEEL_SELECTION | GA_HALT_SELECTION |
                    MIU_AND_LAMDA_SELECTION | MIU_PLUS_LAMDA_SELECTION |
                    ES_HALT_SELECTION
Mutation         -> ONE_PT_MUTATATION | TWO_PT_MUTATION | N_PT_MUTATION |
                    GA_HALT_MUTATION | CEP_MUTATION | FEP_MUTATION |
                    ES_HALT_MUTATION
Crossover        -> ONE_PT_CROSSOVER | TWO_PT_CROSSOVER | N_PT_CROSSOVER |
                    GA_HALT_CROSSOVER | INTERMEDIATE_CROSSOVER |
                    UNIFORMCROSSOVER | ES_HALT_CROSSOVER
Parameters       -> Parameters , Parameter | ε
Parameter        -> RATIO_M | RATIO_C | STDV | EUCLIDEAN | LINEAR_ENTROPY |
                    ROSCA_ENTROPY | GAUSSIAN_ENTROPY | FITPRO_ENTROPY |
                    CLUSTER_ENTROPY
Number           -> integer
Double           -> double
FileName         -> string
```

6. References

Aho, A., Lam, M., Sethi, R. & Ullman J. (2007). *Compilers: Principles, Techniques, and Tools*, Addison Wesley.

Arabas, J., Michalewicz, Z. & Mulawka, J. (1994). GAVaPS - a Genetic Algorithm with Varying Population Size. *The 1st International Conference on Evolutionary Computation*, pp. 73-78

Bäck, T. & Schwefel, H. P. (1995). Evolution Strategies I: Variants and Their Computational Implementation. *Genetic Algorithms in Engineering and Computer Science*. John Wiley & Sons

Bäck, T. & Schütz, M. (1996). Intelligent Mutation Rate Control in Canonical Genetic Algorithms. *Foundations of Intelligent Systems*, pp. 158–167.

Bäck, T., Eiben, A. & van der Vaart., N. A. L. (2000). An Empirical Study on GAs Without Parameters. *In Parallel Problem Solving from Nature VI*, pp. 315–324.

Brooks, F. P. (1987). No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19.

Črepinšek, M., Mernik, M. & Liu, S.-H. (In Press). Analysis of Exploration and Exploitation in Evolutionary Algorithms by Ancestry Trees. *International Journal of Innovative Computing and Applications*.

Dower, S. & Woodward, C. (2010). Evolutionary System Definition Language. Swinburne University of Technology, Tech. Rep. TR/CIS/2010/1

Eiben, A. E., Hinterding, R. & Michalewicz, Z. (1999). Parameter Control in Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2): 124-141.

Eiben, A. E., Marchiori, E. &Valkó, V. A. (2004). Evolutionary Algorithms with On-the-Fly Population Size Adjustment. *Parallel Problem Solving from Nature*, pp. 41-50.

Fogarty, T. C. (1989). Varying the Probability of Mutation in the Genetic Algorithm. *The 3rd International. Conference on Genetic Algorithms*, pp. 104-109

Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Addison Wesley.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns*, Addison-Wesley

Ghosh, A., Tsutsui, S. & Tanaka, H. (1996). Individual Aging in Genetic Algorithms. *Australian New Zealand Conference on Intelligent Information Systems*, pp. 276–279

Gray, J., Fisher, K., Consel, C., Karsai, G., Mernik, M. & Tolvanen, J.-P. (2008). DSLs: the Good, the Bad, and the Ugly. *The 23rd ACM Conference on Object-Oriented Programming Systems Languages and Applications*, pp. 791-794

Grefenstette, J. J. (1986). Optimization of Control Parameters for Genetic Algorithms. IEEE Transaction on Systems, Man & Cybernetics, SMC-16(1): 122-128

Harik, G. & Lobo, F. (1999). A Parameter-less Genetic Algorithm. Technical Report IlliGAL 9900, University of Illinois at Urban-Champaign

Hesser, J. & Männer, R. (1991). Toward an Optimal Mutation Probability for Genetic Algorithms. 1st Conference of Parallel Problem Solving from Nature, pp. 23–32.

Herrera, F. & Lozano, M. (1996). Adaptation of Genetic Algorithm Parameters Based on Fuzzy Logic Controllers. *Genetic Algorithms and Soft Computing*, pp. 95-125

Hudson, S. E. (2010). CUP LALR Parser Generator for Java. http://www2.cs.tum.edu/projects/cup/

Kang, K., Cohen, S., Hess, J., Novak, W. & Peterson., S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University

Keijzer, M., Merelo, J. J., Romero, G. & Schoenauer, M. (2002). Evolving Objects: A General Purpose Evolutionary Computation Library. *Artificial Evolution*, 2310: 829-888.

Kennedy, J. & Eberhart, R.C. (2001). *Swarm Intelligence*. Morgan Kaufmann

Klein, G. (2010). JFlex User's Manual. http://jflex.de/manual.html

Lincke, R., Lundberg, J. & Löwe, W. (2008). Comparing Software Metrics Tools. *International Symposium on Software Testing and Analysis*, pp. 131-142.

Liu, S.-H. (2010). PPCea Web Page. http://www.zimmer.csufresno.edu/~shliu/research/PPCea.html

Liu, S.-H., Mernik, M. & Bryant, B. R. (2004). Parameter Control in Evolutionary Algorithms by Domain-Specific Scripting Language PPCea. *The 1st International Conference on Bioinspired Optimization Methods and their Applications,* pp. 41-50.

Liu, S.-H., Mernik, M. & Bryant, B. R. (2009). To Explore or to Exploit: An Entropy-Driven Approach for Evolutionary Algorithms. *International Journal of Knowledge-based and Intelligent Engineering Systems 13(3-4): 185-206.*

Luke., S. et al. (2010). ECJ: A Java-based Evolutionary Computation Research System. http://cs.gmu.edu/~eclab/projects/ecj/

Meyer, B. (2000). Object-Oriented Software Construction. Prentice Hall.

Michalewicz, Z. (1996). Genetic Algorithm + Data Structures = Evolution Programs. Springer

Mernik, M., Heering, J. & Sloane, A. (2005). When and How to Develop Domain-Specific Languages, *ACM Computing Surveys*, 37(4): 316-344.

Sauer, F. (2010). Eclipse Metrics plug-in. http://sourceforge.net/projects/metrics/

Schach, S. (2010). Object-Oriented and Classical Software Engineering, McGraw Hill.

Smith, R. & Smuda, E. (1995). Adaptively Resizing Populations: An Algorithm, Analysis, and First Results. *Complex Systems*, 1(9): 47-72.

Storn, R. & Price, K. (1997). Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous *Spaces. Journal of Global Optimization,* 11: 341–359.

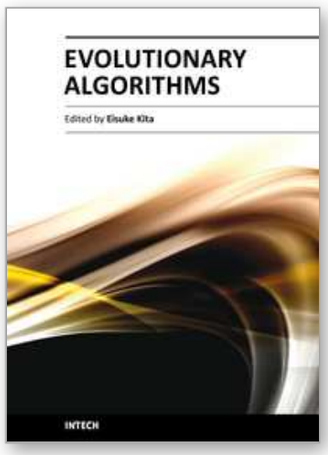
Tsutsui, S., Ghosh, A., Corne, D. & Fujimoto, Y. (1997). A Real Coded Genetic Algorithm with an Explorer and an Exploiter Populations. *The 7th International Conference on Genetic Algorithms*, pp. 238–245.

Ursem, R. (2002). Diversity-Guided Evolutionary Algorithms. *Parallel Problem Solving from Nature* VII, LNCS 2439: pp. 462-471

Veenhuis, C., Franke, K. & Köppen, M. (2000). A Semantic Model for Evolutionary Computation. *6th International Conference on Soft Computing*.

Wu, X., Bryant, B. R., Gray, J. & Mernik, M. (2010). Component-Based LR Parsing. *Computer Languages, Systems, and Structures*, 36(1): 16-33.

Yao, X., Liu, Y., & Lin, G. (1999). Evolutionary Programming Made Faster. *IEEE Transactions on Evolutionary Computation*, 3(2): 82-102.



Evolutionary Algorithms
Edited by Prof. Eisuke Kita

ISBN 978-953-307-171-8
Hard cover, 584 pages
Publisher InTech
Published online 26, April, 2011
Published in print edition April, 2011

Evolutionary algorithms are successively applied to wide optimization problems in the engineering, marketing, operations research, and social science, such as include scheduling, genetics, material selection, structural design and so on. Apart from mathematical optimization problems, evolutionary algorithms have also been used as an experimental framework within biological evolution and natural selection in the field of artificial life.

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen